15-122: Principles of Imperative Computation

Recitation 21b — Josh Zimmerman

Function pointers

Think back to the memory layout recitation. Remember that in the layout of memory there is a section for code.

In C, we can get the address of anything, including things in this code section.

So, we can get pointers to functions, normally referred to as *function pointers*.

Let's consider the following code snippet:

```
1 int square(int a) {
2    return a * a;
3 }
4 int add1(int a) {
5    return a + 1;
6 }
7 void map(int (*f)(int k), int A[], size_t n) {
8    for (int i = 0; i < n; i++) {
9       A[i] = f(A[i]);
10   }
11 }
```

In this example, we wrote a generic function that applies a given function to every value in an array of `int`s. The function can be anything we want it to be, which gives us more flexibility: instead of writing an "add 1 to all in array" function and a "square all in array" function, we can just write the `map` function and pass in a function pointer.

The syntax for function pointers is a bit ugly, but involves the idea of *definition as use*. In C, we declare pointers integers in a way that resembles the way we use them. We say `*x` to use an int pointer, so we declare the variable like this:

```
1 int *x;
```

Similarly, because we *use* a function pointer `f` by writing `(*f)(k)`, returning an `int`, where `k` is an integer expression, we declare the function pointer variable by writing

```
1 int (*f)(int k);
```

It's possible to use `typedef`s to make it easier to deal with. (A handy way to remember `typedef` syntax: if you remove the `typedef` from `typedef foo bar;` then you're declaring a variable named `bar` of type `foo`. For instance, when you do `typedef void *elem;` if you remove the `typedef`, you're declaring an `void*` named `elem`.) We can use function pointers to write things like generic searches and sorts:

```
1 typedef void* elem;
2 typedef int (*compare_fun)(elem x, elem y);
3 int linsearch_min(elem *A, int lower, int upper, compare_fun elem_compare) {
4    REQUIRES(A != NULL && 0 <= lower && lower < upper && elem_compare != NULL);
5    int min_idx = lower;
6    for (int i = lower + 1; i < upper; i++)
7       if ((*elem_compare)(A[i], A[min_idx]) < 0)
8          min_idx = i;
9    return min_idx;
10 }
```

How can we instantiate this to print out the minimum argument passed to a C function?

```
1  int compare_strings(elem x, elem y) {
2     return strcmp((char*)x, (char*)y);
3  }
4
5  int main(int argc, char **argv) {
6     if (argc < 2) {
7        fprintf(stderr, "Not enough arguments\n");
8        return 1;
9     }
10
11    int min = ____________________________________________________________________________;
12
13    printf("The lowest-valued command line argument was %s\n", argv[min]);
14    return 0;
15 }
```

Hashtables

The hashtables and other data structures you will use for Lights Out have the property that they are *generic* – they use void pointers to represent the client's types, and the function pointers that make up the client interface are passed to the client along with the `ht_new` function.

```
1  typedef void *ht_elem; // NULL vs. non-NULL is significant
2  typedef void *ht_key; // NULL vs. non-NULL is not significant
3
4  typedef struct ht_header* ht;
5
6  ht ht_new(size_t capacity,            // Must be > 0
7            ht_key (*elem_key)(ht_elem e), // Must not be NULL
8            bool (*key_equal)(ht_key k1, ht_key k2), // Must not be NULL
9            size_t (*key_hash)(ht_key k),  // Must not be NULL
10           void (*elem_free)(ht_elem e)); // May be NULL
11
12 /* ht_insert(H,e) returns kicked-out element with key of e, if it exists */
13 ht_elem ht_insert(ht H, ht_elem e);
14
15 /* ht_lookup(H,k) returns NULL if no element with key k exists */
16 ht_elem ht_lookup(ht H, ht_key k);
17
18 void ht_free(ht H);
```

1) If we have a `struct wcount` with two fields, a string `word` and an unsigned int `count`, how would we instantiate a hashtable in order to map from words to counts?

2) If we have a `struct twoints` with two fields, an unsigned int `key` and an unsigned int `value`, how would we instantiate a hashtable as a map from keys to values? (You'll want to use the address-of operator to get a pointer to `S->key`, and then cast that `int*` to a `ht_key`.)